Chapter 1

Variant's generation

RQ1. To what extent it is possible to artificially create variants for WebAssembly programs?

This chapter investigates whether we can artificially create program variants through semantically equivalent code transformations. We propose a framework to generate program variants functionally equivalent to their original. We introduce the retargeting of a superoptimizer, using its exhaustive search strategy to provide semantically equivalent code transformations. The presented methodology and transformation tool, CROW, are contributions to this thesis. We evaluate the usage of CROW on two corpora of open-source and nature diverse programs.

1.1 CROW

This section describes CROW, a tool tailored to create semantically equivalent variants out of a single program, either C/C++ code or LLVM bitcode. We assume that the WebAssembly programs are generated through the LLVM compilation pipeline to implement CROW. This assumption is supported by the work of Lehman et al. []; the fact that LLVM-based compilers are the most popular compilers to build WebAssembly programs [**?**] and the availability of source code (typically C/C++; and LLVM for WebAssembly) that provides a structure to perform code analysis and produce code replacements that is richer than the binary code. CROW is part of the contributions of this thesis. In Figure 1.1, we describe the workflow of CROW to create program variants.

Figure 1.1 highlights the main two stages of the CROW's workflow, *exploration* and *combining*. The workflow starts by compiling the input program into the LLVM bitcode using clang from the source code. During the *exploration* stage, CROW takes an LLVM bitcode and, for its code blocks, produces a collection of code replacements that are functionally equivalent to the original program. In the

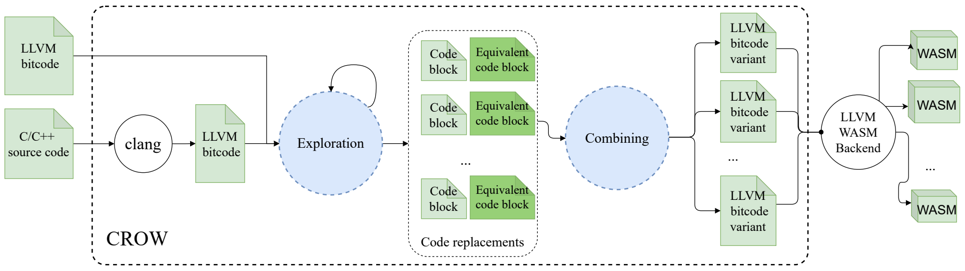

Figure 1.1: CROW workflow to generate program variants. CROW takes C/C++ source codes or LLVM bitcodes to look for code blocks that can be replaced by semantically equivalent code and generates program variants by combining them.

following, we enunciate the definitions we use along with this work for a code block, functional equivalence, and code replacement.

Definition 1 *Block (based on Aho et al. [?]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.*

Definition 2 *Functional equivalence modulo program state (based on Le et al. [?]): Let B_1 and B_2 be two code blocks according to Definition 1. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.*

Definition 3 *Code replacement: Let P be a program and T a pair of code blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both functionally equivalent as defined in Definition 2. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.*

We implement the *exploration* stage by retargeting a superoptimizer for LLVM, using its subset of the LLVM intermediate representation. CROW operates at the code block level, taking them from the functions defined inside the input LLVM bitcode module. In addition, the retargeted superoptimizer is in charge of finding the potential places in the original code blocks where a replacement can be applied. Finally, we use the enumerative synthesis strategy of the retargeted superoptimizer to generate code replacements. The code replacements generated through synthesis are verified, according to Definition 2, by internally using a theorem prover.

Moreover, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly for the sake of reducing the searching space for equivalent program variants. Besides, we disable all optimizations in the WebAssembly LLVM backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

In the *combining* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants, selecting and merging the code replacements. Then for each combination, a variant bitcode is compiled into a WebAssembly binary if requested. Finally, CROW generates the variants from all possible combinations of code replacements as the power set of all code replacements.

1.1.1 Example

Let us illustrate how CROW works with the simple example code in Listing 1.1. The `f` function calculates the value of $2*x+x$ where `x` is the input for the function. CROW compiles this source code and generates the intermediate LLVM bitcode in the left most part of Listing 1.2. CROW potentially finds two code blocks to look for variants, as the right-most part of Listing 1.2 shows.

Listing 1.1: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

Listing 1.2: LLVM's intermediate representation program, its extracted code blocks and replacement candidates. Gray highlighted lines represent original code block, green for candidate code replacements.

```
define i32 @f(i32) {

    code block 2
    code block 1
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

ret i32 %3
}

define i32 @main() {
%1 = tail call i32 @f(i32
     10)
ret i32 %1
}
```

```
Replacement candidates for
       code_block_1

%2 = mul nsw i32 %0,2

%2 = add nsw i32 %0,%0

%2 = shl nsw i32 %0, 1:i32
```

```
Replacement candidates for
       code_block_2

%3 = add nsw i32 %0,%2

%3 = mul nsw %0, 3:i32
```

CROW, in the exploration stage detects 2 code blocks, `code_block_1` and `code_block_2` as the enclosing boxes in the left most part of Listing 1.2 show. CROW synthesizes $2 + 1$ candidate code replacements for each code block respectively as the green highlighted lines show in the right most parts of Listing 1.2. The baseline strategy of CROW is to generate variants out of all possible combinations of the candidate code replacements, *i.e.,* uses the power set of all candidate code replacements.

Listing 1.3: Candidate code replacements combination. Orange highlighted code illustrate replacement candidate overlapping.

```
%2 = mul nsw i32 %0,2                   %2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2                  %3 = mul nsw %0, 3:i32

%2 = add nsw i32 %0,%0                  %2 = add nsw i32 %0,%0
%3 = add nsw i32 %0,%2                  %3 = mul nsw %0, 3:i32

%2 = shl nsw i32 %0, 1:i32              %2 = shl nsw i32 %0, 1:i32
%3 = add nsw i32 %0,%2                  %3 = mul nsw %0, 3:i32
```

In the example, the power set is the cartesian product of the found candidate code replacements for each code block, including the original ones, as Listing 1.3 shows. The power set size results in 6 potential function variants. Yet, the generation stage would eventually generate 4 variants from the original program. CROW generated 4 statically different Wasm files, as Listing 1.4 illustrates. This gap between the potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composed into one. In other words, if the replaced code removes other code blocks, all possible combinations having it will be in the end the same program. In the example case, replacing `code_block_2` by `mul nsw %0, 3`, turns `code_block_1` into dead code, thus, later replacements generate the same program variants. The rightmost part of Listing 1.3 illustrates how for three different combinations, CROW produces the same variant. We call this phenomenon an overlapping.

One might think that a reasonable heuristic could be implemented to avoid such overlapping cases. Instead, we have found it easier and faster to generate the variants with the combination of the replacement and check their uniqueness after the program variant is compiled. This prevents us from having an expensive checking for overlapping inside the CROW code. Still, this phenomenon calls for later optimizations in future works.

Listing 1.4: Wasm program variants generated from program Listing 1.1.

```
func $f (param i32) (result i32)             func $f (param i32) (result i32)
  local.get 0                                    local.get 0
   i32.const 2                                   i32.const 1
   i32.mul                                       i32.shl
   local.get 0                                   local.get 0
   i32.add                                       i32.add

func $f (param i32) (result i32)             func $f (param i32) (result i32)
   local.get 0                                   local.get 0
   local.get 0                                   i32.const 3
   i32.add                                       i32.mul
   local.get 0
   i32.add
```

1.2 Evaluation

We use CROW, the tool described at section 1.1, to answer RQ1. This section describes the corpora of original programs that we pass to CROW for the sake of variants generation. Besides, we describe our metrics and finalize the section by discussing the results.

1.2.1 Corpora

We answer RQ1 with two corpora of programs appropriate for our experiments. The first corpus, **CROW prime**, is part of the CROW contribution []. The second corpus, **MEWE prime**, is part of the MEWE contribution []. In Table 1.1 we summarize the selection criteria, and we mention each corpus properties. With both corpora we evaluate CROW with a total of $303 + 2527$ functions.

1.2.2 Metric

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of unique variants generated by CROW for each original function. We compare the WebAssembly program and its variant using the `md5` hash of each function byte stream as a metric for uniqueness.

1.2.3 Setup

CROW's workflow synthesizes program variants with an enumerative strategy. All possible programs that can be generated for a given language (LLVM in the case) are constructed and verified for equivalence. There are two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. On the other hand, in our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer).

These two configuration parameters allow the user to find a trade-off between the number of variants that are synthesized and the time taken to produce them. In Table 1.2 we listed the configuration for both corpora. For the current evaluation, given the size of the corpus, we set the exploration time to 1 hour maximum per function for **CROW PRIME**. In the case of **MEWE prime**, we set the timeout to 5 minutes per function in the exploration stage. We set all 60 supported instructions in CROW for both **CROW prime** and **MEWE primer** corpora.

Corpus name	Selection criteria	Corpus Description
CROW prime	We take programs from the Rosetta Code project[1]. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. We then apply a number of filters: the programs should successfully compile, they should not require user inputs, the programs should terminate and should not provide in non-deterministic results. The result of the filtering is a corpus of 303 C programs	All programs have a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.
MEWE prime	We select two mature and typical edge-cloud computing projects for this corpus. The projects are selected based on their suitability for diversity synthesis with CROW, *i.e.,* the projects should have the ability to collect their modules in LLVM intermediate representation The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust.	The evaluated projects contain in total 2527 functions, 687 for libdosium and 1840 for qrcode-rust. The functions range between 10 ad 127700 lines of code.

Table 1.1: Corpora description. The table is composed by the name of the corpus, the selection criteria and the stats the programs in each corpus.

CORPUS	Exploration timeout	Max. instructions
CROW prime	1h	60
MEWE prime	5m	60

Table 1.2: CROW tweaking for variants generation. The table is composed by the name of the corpus, the timeout parameter and the count of allowed instructions during the synthesis process.

1.3 Results

We summarize the results in Table 1.3. CROW produces at least one unique program variant for 239/303 single function programs for **CROW prime** with 1h for timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of **MEWE prime**, CROW produces variants for 219/2527 functions with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants.

CORPUS	#Functions	# Diversified	# NonDiversified	# Variants
CROW prime	303	**239**	64	1906
MEWE prime	2527	219	2308	**6563**

Table 1.3: General diversification results. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the number of non-diversified functions and the cumulative number of variants.

1.3.1 Challenges for automatic diversification

CROW generates variants for functions in both corpora. However, we have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions, as it can be appreciated in Table 1.3. CROW successfully diversified approx. 79 % and 8.67 % of the original functions for **CROW prime** and **MEWE prime** respectively. On the other hand, CROW generated more variants for **MEWE prime**, 6563 program variants for 219 diversified programs. Not surprisingly, setting the timeout affects the capacity of CROW for diversification. On the other hand, a low timeout for exploration gives CROW more power to combine code replacements. This can be appreciated in the last column of the table, where for a lower number of diversified functions, CROW created, overall, more variants.

Moreover, we look at the cases that yield a few variants per function. There is no direct correlation between the number of identified codes for replacement and the number of unique variants. Therefore, we manually analyze programs that include many potential places for replacements, for which CROW generates few or no variants. We identify two main challenges for diversification.

1) Constant computation We have observed that Souper searches for a constant replacement for more than 45% of the blocks of each function while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because CROW is oblivious to a memory model.

2) Combination computation The overlap between code blocks, mentioned in subsection 1.1.1, is a second factor that limits the number of unique variants. CROW can generate a high number of variants, but not all replacement combinations are necessarily unique.

1.3.2 Properties for large diversification using CROW

We manually analyzed the programs that yield more than 100 unique variants to study the critical properties of programs leveraging a high number of variants. This reveals one key reason that favors many unique variants: the programs include bounded loops. In these cases, CROW synthesizes variants for the loops by replacing them with a constant if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which CROW can successfully infer.

A second key factor for synthesizing many variants relates to the presence of arithmetic. Souper, the synthesis engine used by CROW, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, CROW generates unique variants by replacing multiplications with additions or shift left instructions (Listing 1.5). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 1.6). Besides, CROW can use overflow and underflow of integers to produce variants (Listing 1.7), using the intrinsics of the underlying computation model.

Listing 1.5: Diversification through arithmetic expression replacement.

```
local.get 0      local.get 0
i32.const 2      i32.const 1
i32.mul          i32.shl
```

Listing 1.6: Diversification through inversion of comparison operations.

```
local.get 0      i32.const 11
i32.const 10     local.get 0
i32.gt_s         i32.le_s
```

Listing 1.7: Diversification through overflow of integer operands.

```
i32.const 2      i32.const 2
i32.mul          i32.mul
                 i32.const -2147483647
                 i32.mul
```

1.3.3 Variant properties

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the diversified programs with their variants. The ratio of size change between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for **CROW prime** and **MEWE prime**. This limited impact on the binary size of the variants is good news because they are meant to save bandwidth when they become assets to distribute over the network.

1.4 Conclusions

The proposed methodology can generate program variants that are syntactically different from their original versions. We have shown that CROW generates diversity among the binary code variants using semantically equivalent code transformations. We identified the properties that original programs should have to provide a handful number of variants. Besides, we enumerated the challenges faced to provide automatic diversification by retargeting a superoptimizer.

In the next chapter, we evaluate the assessment of the generated variants answering to what extent the artificial programs are different from the original in terms of static difference, execution behavior, and preservation.